Programação Paralela com Troca de Mensagens

Profa Andréa Schwertner Charão

DLSC/CT/UFSM

Sumário

- Modelo de programa MPI
- Comunicação em MPI
  - Comunicadores
  - Mensagens
  - Comunicação ponto-a-ponto
  - Comunicação coletiva

Bibliotecas para troca de mensagens

- Padrão MPI (Message Passing Interface)
  - Implementações: OpenMPI, MPICH, LAM, etc.
- Alternativas: sockets, ZeroMQ, PVM (Parallel Virtual Machine), etc.
- Funcionalidades

processos	Criação, identificação, etc.
comunicação	Ponto-a-ponto, em grupo, global, etc.
sincronização	Barreiras, comunicação síncrona

MPI

- API padrão para troca de mensagens (versões MPI1, MPI2, MPI3)
- Programas escritos em C, C++, Fortran, Python, etc.
- Implementações
  - open-source: OpenMPI, MPICH, LAM, ...
  - proprietárias
- Mais de 120 chamadas
  - diferentes modos de comunicação
  - muitos programas usam menos de 10 chamadas!

Processos em MPI

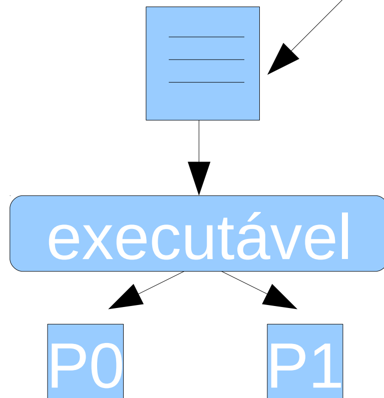
- Criação e execução de processos
  - MPI versão 1
    - ✓ Criação estática de processos: processos devem ser definidos antes da execução e inicializados juntos
    - ✓ Utiliza o modelo de programação SPMD
  - MPI versão 2
    - ✓ Criação dinâmica de processos
    - ✓ Permite utilizar um modelo MPMD

Modelos SPMD e MPMD

- Single Program Multiple Data

seleção de código conforme id. do processo

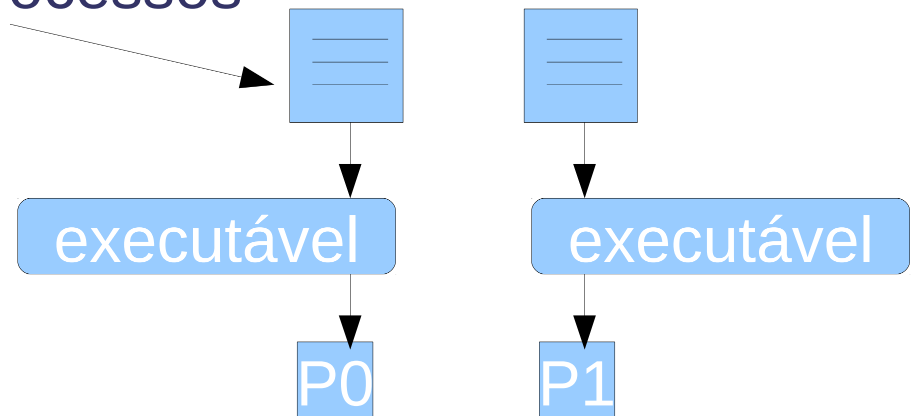
código fonte



- Multiple Program Multiple Data

código que ativa outros processos

código fonte



Modelo SPMD

```
#include "mpi.h"

int main (int argc, char *argv[])
{
     MPI_Init(&argc, &argv);
     .
     .
     MPI_Comm_Rank(MPI_COMM_WORLD, &myrank);
     if (myrank ==0)
        ...;
     else
        ...;
     .
     .
     MPI_Finalize();
}
```

Comunicação em MPI

- Contextos/grupos de comunicação (communicators)
  - **MPI_COMM_WORLD:** contexto/grupo contendo todos os processos lançados
- Processos de um grupo são numerados de 0 a NP-1
- Operações básicas com grupos
  - **MPI_Comm_rank**: id do processo no grupo
  - **MPI_Comm_size**: número de processos no grupo

Mensagens em MPI

- Tamanho (quantidade de dados)
- Tipo do dado (**MPI_INT**, **MPI_DOUBLE**, etc.)
- Identificação de processo (origem, destino, etc.)
  - **MPI_ANY_SOURCE** para recepção de qualquer processo
- Índice (tag)
  - **MPI_ANY_TAG** para recepção de qualquer tag
- Comunicator (ex.: MPI_COMM_WORLD)
- Status (útil principalmente para recepções)

Comunicação em MPI

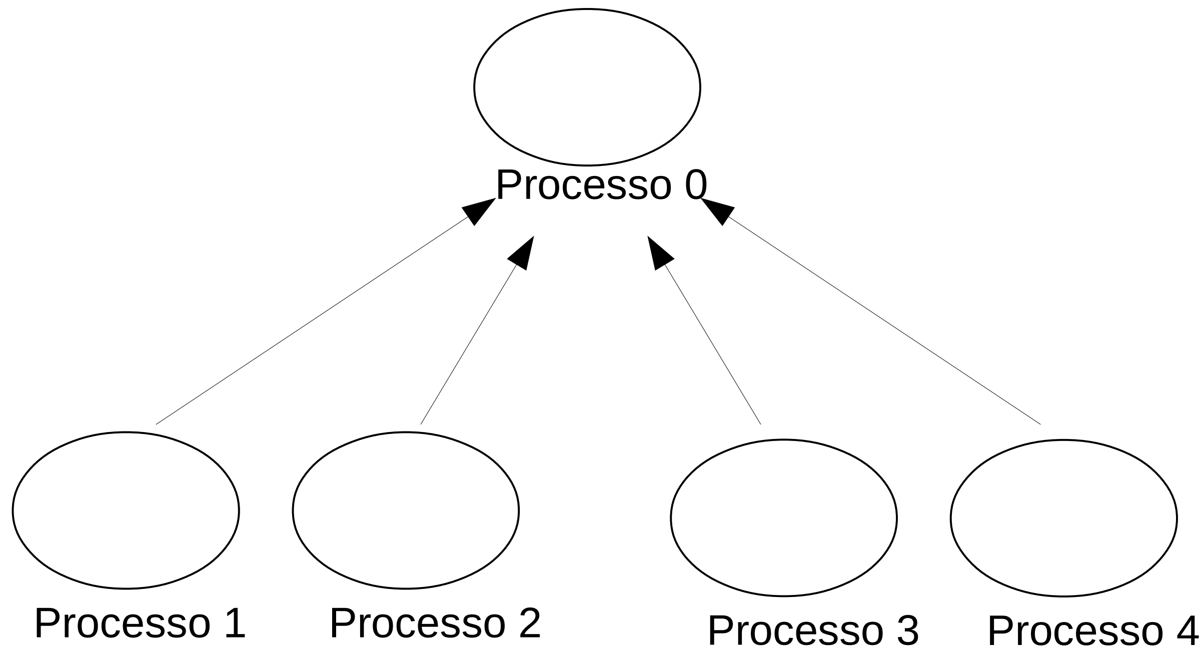
- Comunicação ponto-a-ponto (local)
  - com bloqueio
  - sem bloqueio
  - modos de envio: padrão, bufferizado, síncrono, ready
- Comunicação coletiva (global)
  - processo raiz
  - operações: difusão, redução, distribuição, coleta, etc.

Comunicação ponto-a-ponto

- Com bloqueio
  - **MPI_Send/MPI_Recv**: bloqueiam até que dados da mensagem possam ser utilizados
- Sem bloqueio
  - **MPI_Isend/MPI_Irecv**: envio/recepção com retorno imediato (sem bloqueio)
  - Possuem **MPI_Request** associado
  - **MPI_Wait/MPI_Test**: espera/verifica se operação foi completada

Exemplo: comunicação ponto-a-ponto

- Olá, mundo!
  - Processos 1 a P-1 enviam “Olá”
  - Processo 0 recebe as mensagens



cabeçalho MPI

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;        // "rank" do processo (0 a P-1)
  int p;               // número de processos
  int origem;          // "rank" do processo remetente
  int destino;         // "rank" do processo destinatário
  int tag = 0;         // "etiqueta" da mensagem
  char msg[100];       // a mensagem
  MPI_Status status;   // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;        // "rank" do processo (0 a P-1)
  int p;               // número de processos
  int origem;          // "rank" do processo remetente
  int destino;         // "rank" do processo destinatário
  int tag = 0;         // "etiqueta" da mensagem
  char msg[100];       // a mensagem
  MPI_Status status;   // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

inicialização do ambiente

finalização do ambiente

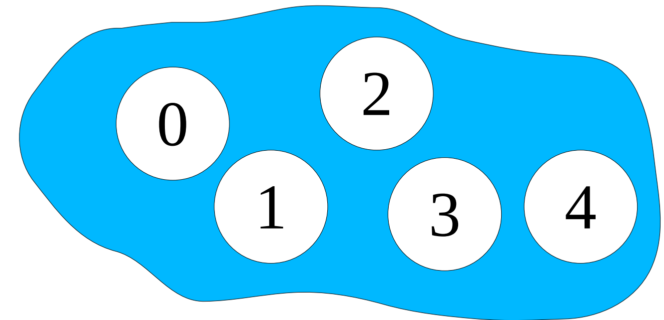
```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;          // "rank" do processo (0 a P-1)
  int p;                 // número de processos
  int origem;            // "rank" do processo remetente
  int destino;           // "rank" do processo destinatário
  int tag = 0;           // "etiqueta" da mensagem
  char msg[100];         // a mensagem
  MPI_Status status;     // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, des
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

"quem sou eu?"



```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;        // "rank" do processo (0 a P-1)
  int p;               // número de processos
  int origem;          // "rank" do processo remetente
  int destino;         // "rank" do processo destinatário
  int tag = 0;         // "etiqueta" da mensagem
  char msg[100];       // a mensagem
  MPI_Status status;   // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

quantos processos fazem parte do “comunicador”?

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;          // "rank" do processo (0 a P-1)
  int p;                 // número de processos
  int origem;            // "rank" do processo remetente
  int destino;           // "rank" do processo destinatário
  int tag = 0;           // "etiqueta" da mensagem
  char msg[100];         // a mensagem
  MPI_Status status;     // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

processos 1 a p-1 executam esse código

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;         // "rank" do processo (0 a P-1)
  int p;                // número de processos
  int origem;           // "rank" do processo remetente
  int destino;          // "rank" do processo destinatário
  int tag = 0;          // "etiqueta" da mensagem
  char msg[100];        // a mensagem
  MPI_Status status;    // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;          // "rank" do processo (0 a P-1)
  int p;                 // número de processos
  int origem;            // "rank" do processo remetente
  int destino;           // "rank" do processo destinatário
  int tag = 0;           // "etiqueta" da mensagem
  char msg[100];         // a mensagem
  MPI_Status status;     // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
                     k" do processo
                     COMM_WORLD, &meu_rank);
                     co de processos
                     COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

envia mensagem

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;        // "rank" do processo (0 a P-1)
  int p;               // número de processos
  int origem;          // "rank" do processo remetente
```

```
int MPI_Send(
     void *buf,                   // ptr para msg
     int count,                   // tamanho da msg
     MPI_Datatype dtype,          // tipo de dados na msg
     int dest,                    // destinatário
     int tag,                     // identificador da msg
     MPI_Comm comm)               // conjunto “comunicador”
```

```
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;        // "rank" do processo (0 a P-1)
  int p;               // número de processos
  int origem;          // "rank" do processo remetente
  int destino;         // "rank" do processo destinatário
  int tag = 0;         // "etiqueta" da mensagem
  char msg[100];       // a mensagem
  MPI_Status status;   // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

processo 0
recebe p-1
mensagens

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;        // "rank" do processo (0 a P-1)
  int p;               // número de processos
  int origem;          // "rank" do processo remetente
  int destino;         // "rank" do processo destinatário
  int tag = 0;         // "etiqueta" da mensagem
  char msg[100];       // a mensagem
```

```
int MPI_Recv(
       void *buf,                   // ptr para msg
       int count,                   // tamanho máximo da msg
       MPI_Datatype dtype,          // tipo de dados na msg
       int source,                  // remetente
       int tag,                     // identificador da msg
       MPI_Comm comm,               // conjunto "comunicador"
       MPI_STATUS *stat)            // status da operação
```

```
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int meu_rank;        // "rank" do processo (0 a P-1)
  int p;               // número de processos
  int origem;          // "rank" do processo remetente
  int destino;         // "rank" do processo destinatário
  int tag = 0;         // "etiqueta" da mensagem
  char msg[100];       // a mensagem
  MPI_Status status;   // "status" de uma operação efetuada

  // MPI_Init deve ser invocado antes de qualquer outra chamada MPI
  MPI_Init(&argc, &argv);
  // Descobre o "rank" do processo
  MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
  // Descobre o número de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (meu_rank != 0) {
    sprintf(msg, "Processo %d disse Ola!", meu_rank);
    // envia mensagem ao processo 0
    destino = 0;
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
  } else {
    for(origem = 1; origem < p; origem++) {
      // recebe P-1 mensagens
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", msg); // mostra mensagem
    }
  }
  MPI_Finalize(); // finaliza MPI
  return 0;
}
```

Comunicação coletiva

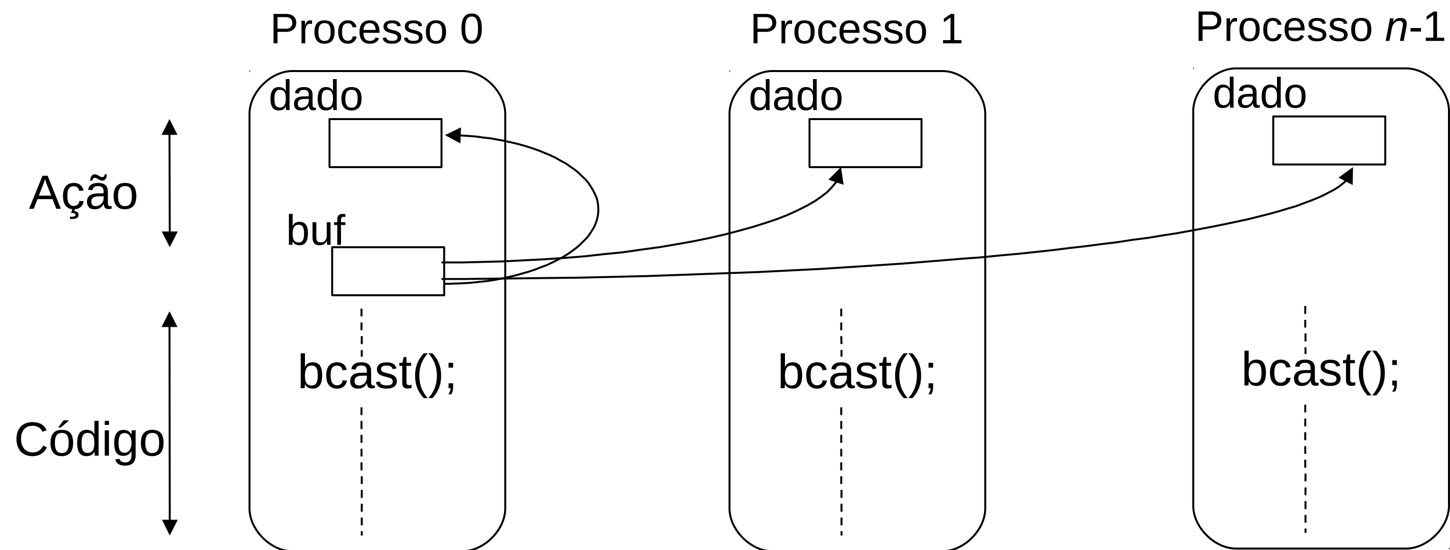
- Realizada em um conjunto de processadores definido por um communicator
- Não usa índices (tags)
- Todos processos usam a mesma função
- **MPI_Bcast**: envio do processo raiz para todos os outros
- **MPI_Gather**: coleta valores para um grupo de processos
- **MPI_Scatter**: espalha buffer de dados em partes para um grupo de processos
- **MPI_Alltoall**: envia dados de todos os processos para todos os processos

Comunicação coletiva

- **MPI_Reduce**: combina valores de todos os processos em um único valor no processo raiz
- **MPI_Allreduce**: combina valores e difunde resultados
- **MPI_Reduce_scatter**: combina valores e espalha resultado

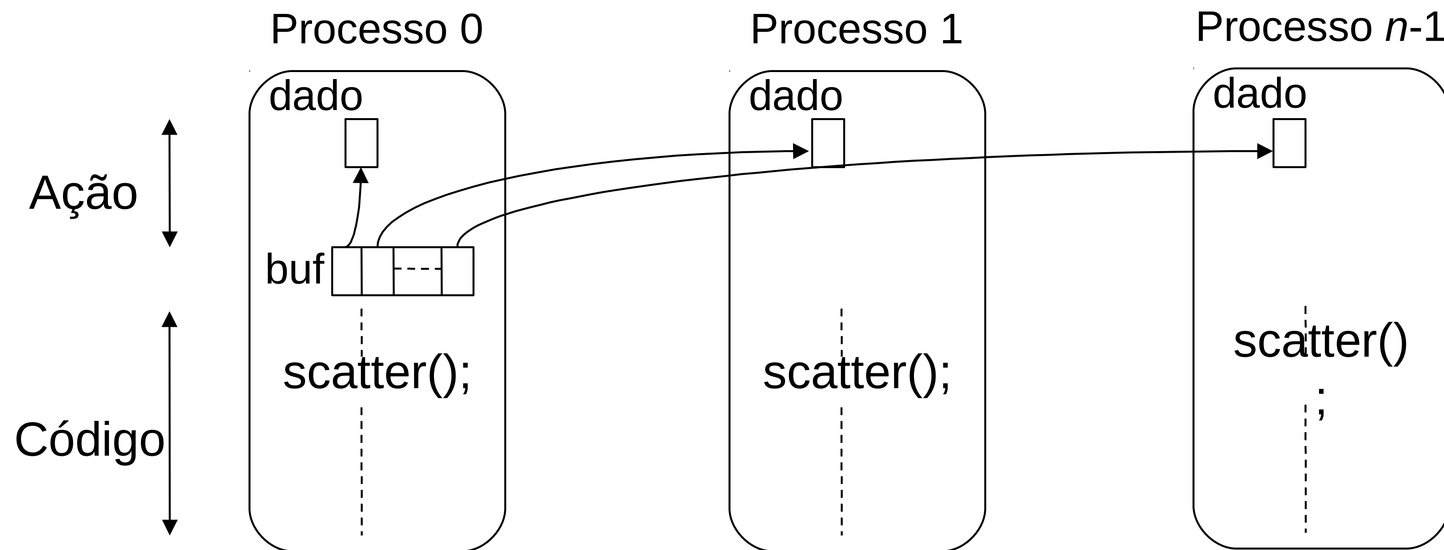
MPI_Bcast

- Envio da mesma mensagem para todos os processos



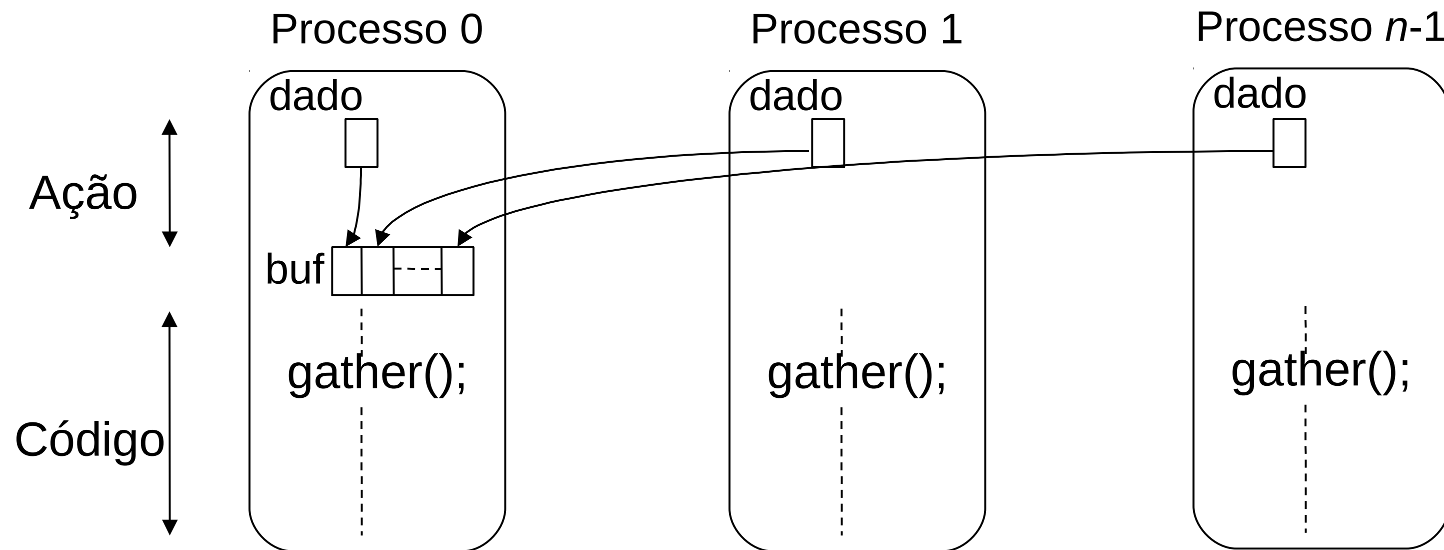
MPI_Scatter

- Envio de cada elemento de uma matriz de dados do processo raiz para outros processos; o conteúdo da i-ésima localização da matriz é enviado para o i-ésimo processo



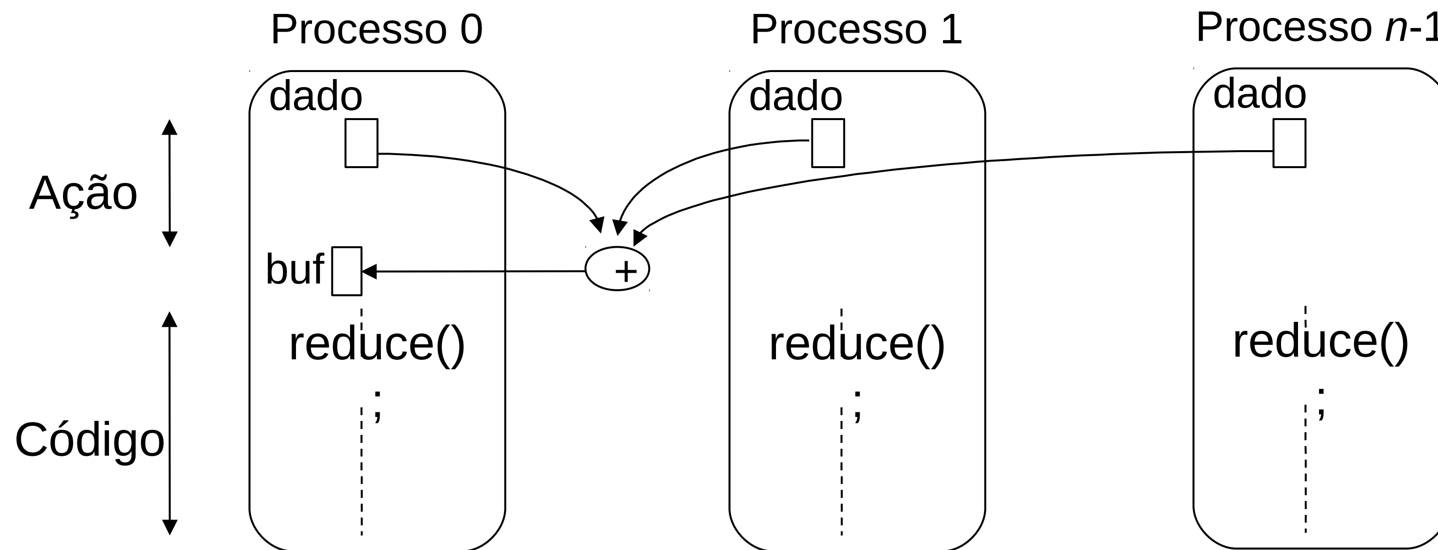
MPI_Gather

- Coleta de dados de um conjunto de processos, com armazenamento no processo raiz



MPI_Reduce

- Operação MPI_Gather combinada com uma operação lógica ou aritmética específica. Ex: valores coletados e somados pelo processo raiz



Sincronização com barreira (barrier)

- Em todos os sistemas de passagem de mensagens existe uma maneira de sincronizar os processos
- **MPI_Barrier()**: processos ficam bloqueados até todos os processos terem atingido essa instrução no programa